Cap.5 Implementarea unui Sistem Expert

5.1. Noțiuni generale despre SE

"Sistemele expert sunt programe concepute pentru a raționa în scopul rezolvării problemelor pentru care în mod obişnuit se cere o expertiză umană considerabilă", Edward Feingenbaum - Standford University

Se pot oferi foarte multe definiții pentru noțiunea de sistem expert, şi de asemenea se pot realiza foarte multe clasificări pornind de la acestea. Cert este faptul că un SE încorporează o bază de cunoştințe şi un motor de inferență şi că acesta utilizează proceduri de inferențe pentru soluționarea unor probleme. S-au dezvoltat strategii eficiente de căutare cum ar fi de exemplu: analizele means-ends utilizate în GPS şi în planificare, strategiile alfa-beta din jocuri, algoritmul A* pentru căutare euristică. Cu toate acestea SE au grad foarte mare de generalitate. Pentru a contura noțiunea de sistem expert putem enumera următoarele idei:

- din punct de vedere conceptual sistemele expert vizează reconstituirea raționamentului uman pe baza expertizei obținută de experți;
- sistemele expert dispun de cunoştințe şi de capacitatea de a desfăşura activități intelectuale umane;
- sunt organizate pentru achiziția şi exploatarea cunoaşterii dintr-un domeniu particular numit domeniul problemei;
- dispun de metode de invocare a cunoaşterii şi exprimarea expertizei comportându-se ca un: "asistent inteligent".
- la nivel de realizare informatică sistemele expert se bazează pe principiul separării cunoaşterii de programul care o tratează;
- sunt capabile să memoreze cunoaşterea ,să stabilească legăturile între cunoştințe şi să infereze concluzii, soluții, recomandări, sfaturi, respectiv cauzele unor fenomene.

Cunoaşterea într-un sistem expert este organizată într-o manieră care separă cunoştințele despre domeniul problemei de alte tipuri de cunoştințe cum sunt cele despre rezolvarea problemei şi cele despre interacțiunea cu utilizatorul. Se evidențiază în funcție de tipul de cunoştințe utilizate trei componente principale:

- *Baza de cunoştințe*, pentru stocarea tuturor pieselor de cunoaştere specifice unui domeniu aplicativ, creată şi organizată pentru satisfacerea obiectivului învățării.
- *Motorul de inferență,* un program care conține cunoaşterea de control, procedurală sau operatorie, care exploatează baza de cunoştințe şi este destinat satisfacerii combinării şi înlănțuirii cunoaşterii.
- *Interfața de dialog cu utilizatorii* care dispune şi de un limbaj de exprimare a cunoaşterii achiziționată de la experții umani.

Aceste componente nu acționează izolat de celelalte componente ale mediului exterior în care SE este instalat. El poate fi apelat de către alte programe externe sau poate trimite rezultate către alte programe; sunt interactive în sensul că

interfața lor oferă mijloace de comunicații cu utilizatorii umani, asigură obținerea datelor şi informațiilor de la senzori speciali, de la sisteme de gestiune a bazelor de date, de la sisteme de calcul tabelar sau chiar de la programul care gestionează fişiere clasice, pot crea şi actualiza baze de date, pot imprima rapoarte şi pot controla cele mai diverse dispozitive şi sisteme.

Baza de cunoştințe conține informații preluate de la experții umani în legătură cu domeniul problemei, care descriu situații evidente, fapte reale sau ipotetice, precum şi euristici. Modulul de achiziție al cunoaşterii ajută utilizatorul expert să introducă cunoştințe într-o formă recunoscută de către sistem şi să actualizeze baza de cunoştințe. Anumite sisteme expert folosesc o bază de date relațională în care sunt memorate explicit evenimente, fapte, obiecte pentru o mai bună flexibilitate a memorării şi regăsirii. Atunci când cunoaşterea este memorată sub forma de reguli de producție, baza de cunoştințe conține două componente: baza de fapte şi baza de reguli.

Motorul de inferență este un program sau chiar un circuit integrat microprogramat care dispune de mecanisme inferențiale generale pentru prelucrarea cunoştințelor cu raționamente din cele mai diverse, determinând modificarea cunoaşterii în scopul soluționării problemei.

Motorul de inferență are două componente de bază:

a) sistemul de administrare a bazei de cunoştințe;

b) procesorul de interfețe simbolic.

Sistemul de administrare a bazei de cunoştințe efectuează operații de organizare automată, control şi actualizare a cunoştințelor, inițiază căutări pentru controlul relevanței pe linii de raționament pe care lucrează procesorul de inferențe simbolic.

Termenul de euristică derivă din grecescul heuristikein cu semnificația de a inventa, a descoperi. Gândirea euristică nu se produce în mod direct, ea implică judecată, căutare, reînvățare, evaluare şi exploatare, iar cunoaşterea obținută prin această succesiune de procese poate la rândul ei modifica procesul de căutare în vederea soluționării problemelor. Euristicile sunt utilizate adesea de către om când soluționează o problemă sau ia decizii. Ele se bazează pe experiența trecută sau pe cunoaşterea acumulată. Procesele inferențiale implica utilizarea raționamentului. Strategiile de raționament utilizate în mod obişnuit în soluționarea problemelor includ deducția, inducția şi abducția.

Waterman arată că sistemele expert trebuie să aibă abia patru caracteristici de bază: *expertiza, raționamentul simbolic, profunzimea* şi *autocunoaşterea.*

Joseph Giarratano şi Gary Rilei prezintă în alt mod caracteristicile sistemelor expert: înaltă *performanță* în calitatea concluziilor, *timp de răspuns adecvat*, bună *fiabilitate*, capacitatea de a exprima raționamente , *flexibilitatea*.

Frank Puppe de la Universitatea din Karlusruhe arată că listarea proprietăților necesare sistemelor expert este de fapt un alt mod de a le caracteriza. Aceste proprietăți sunt: *transparența* care reprezintă puterea de a explica soluția oferită prin indicarea cunoştințelor utilizate in raționament, *flexibilitatea* care

reprezintă posibilitatea adăugării, schimbării şi eliminării cunoştinţelor, *competenţa* caracterizată prin capacitatea de a soluţiona probleme pentru expertul uman.

Toate tipurile de cunoaştere permit experţilor să ia decizii mai rapide şi mai bune decât neexperţii în soluţionarea problemelor complexe. În legătură cu expertiza trebuie menţionat că ea este uzual asociată cu un înalt grad de inteligenţă, cu mari cantităţi de cunoştinţe.

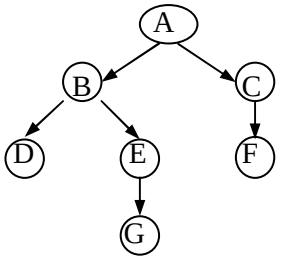
Primele programe inteligente care şi-au dovedit superioritatea într-un domeniu specific al cunoaşterii în raport cu metodele generale de rezolvare a problemelor au fost MACSYMA, un simplificator de expresii algebrice şi DENDRAL care identifica moleculele pe baza spectogramelor. Cel mai important este însă MYCIN pentru diagnoza şi tratamentul infecţiilor sanguine.

Dat fiind faptul că limbajul CLIPS este un generator de sisteme expert, ne propunem exemplificarea unor metode simple de realizare a unui SE.

5.2. Construirea unui arbore sau a unui graf

Cele mai simple sisteme expert sunt cele prin care utilizatorului i se pune o întrebare şi în funcţie de răspunsul acestuia, domeniul problemei este divizat în mai multe subdomenii. De obicei variantele de răspuns la o astfel de întrebare sunt de tip afirmativ sau negativ şi în acest caz se aplică principiul *dihotomiei* (tăierii în două). Se pot implementa şi SE în care utilizatorul poate dispune de mai multe variante de răspuns pentru o întrebare şi în această situaţie domeniul problemei se împarte în mai mult de două subdomenii.

Structura cea mai des folosită pentru reprezentarea problemei este cea arborescentă, dar se pot implementa şi SE de tip graf (vor fi exemplificate şi astfel de sisteme expert). Putem reprezenta un nod dintr-un arbore folosind un fapt ordonat *(nod A B C)*, unde primul element al listei reprezintă numele nodului părinte, iar celelalte elemente reprezintă *fiii* acestuia. Se pot folosi şi fapte deftemplate *(nod (nume A) (fii B C))*, prin definirea unui şablon în care slotul fii este de tip multicâmp *(deftemplate nod (slot nume) (multislot fii))*. Folosind pentru început faptele ordonate putem defini foarte uşor următorul arbore:




```
(deffacts noduri_arbore
        (nod A B C)
        (nod B D E)
        (nod C F)
        (nod E G)
        (nod G) )
```

În construcţia deffacts de mai sus se pot omite nodurile frunză (cum ar fi nodul G), deoarece acestea nu au copii şi de existenţa acestora putem afla urmărind lista fiilor părinţilor acestora (respectiv nodul E). Utilitatea asertării unui fapt *nod* constă practic în enumerarea copiilor acestuia.

Dacă dorim să aflăm care sunt nodurile terminale (frunzele), respectiv rădăcina arborelui se pot defini două reguli în care se pune condiția ca acel nod să nu aibă un tată, respectiv un fiu.

```
(defrule radacina_arbore
        (nod  ?r  $?)
        (not (exists (nod  ?   $?  ?r  $?) ))
        =>
        (assert (level 1 ?r) (level 2 ) (depth 1))
        (printout t "Nodul " ?r " este radacina arbore" crlf)
)
(defrule frunze_arbore
        (nod  ?    $?  ?f  $?)
        (not (nod  ?f   ?  $?))
        =>
        (printout t "Nodul " ?f " este frunza arbore" crlf)
)
```

Folosirea unui element condițional *(not ...)* este echivalentă cu folosirea lui *(not (exists...))* în cazul în care în interiorul acestora se află un singur CE. De reținut că folosind CE *(not (exists ...))* se poate nega existența mai multor fapte. Se observă folosirea variabilelor libere în interiorul pattern-urilor pentru a înlocui valorile ce nu ne interesează. Prin pattern-ul *(not (nod ? $? ?r $?))* se pune condiția "să nu existe un nod care să aibă nodul ?r copil", iar prin *(nod ? $? ?f $?) (not (nod ?f ? $?))* se selectează un fiu al unui nod oarecare şi se pune condiția ca acesta să nu aibă nici un fiu. Folosirea lui *(not (nod ?f $?))* nu este de dorit, deoarece ne forțează să nu introducem în lista de fapte decât nodurile care au copii. Dacă în lista de fapte ar exista şi noduri terminale acestea nu ar fi afişate de regulă. Introducerea unui test suplimentar *(not (nod ?f $?val&:(neq 0 (length$ $?val))))* ar remedia însă această problemă.

Vom realiza şi un set de reguli prin care putem determina nivelele arborelui şi respectiv adâncimea acestuia. În regula *rădăcina_arbore* prin asertarea faptelor (level 2) şi (depth 2) se pregăteşte aprinderea acestor reguli.

```
(defrule search
        (level ?no $? ?tata $?)                          ;nivel anterior
        (depth =(+ ?no 1))
        (nod ?tata $?fii&:(neq 0 (length $?fii)) )
        =>
        (assert (adauga $?fii))
)
(defrule add_to_level                          ;??? prioritate mai mică
        ?f1 <- (adauga $?fii)
        ?f2 <- (level ?no $?val)
        (depth ?no)
        =>
        (retract ?f1 ?f2)
        (assert (level ?no $?val $?fii))
)
```

```
(defrule next_level
        (declare (salience -10))
        ?f <- (depth ?no)
        (not (level ?no ))
        =>
        (retract ?f)
        (assert (depth (+ ?no 1)) (level (+ ?no 1)))
)
```

Pentru a se evita realizarea unei bucle infinite într-o regulă în care se şterge şi se reasertează faptul *level* au fost definite două reguli *search* şi *add_to_level*. Pentru a se trecere la un alt nivel trebuie îndeplinită condiția de a se fi găsit cel puțin un element pentru nivelul actual. Dacă faptul *level* nu conține decât numărul nivelului, atunci CE *(not (level ?no))* nu este satisfăcut şi regula *next_level* nu se aprinde. Faptul *depth* va conține întotdeauna valoarea ultimului nivel pentru care s-a căutat un element, şi deci în final pentru a afişa adâncimea arborelui este necesară decrementarea acestei valori cu unu.

```
(defrule print_depth
        (declare (salience -20))
        (depth ?no)
        =>
        (printout t "Adancimea arborelui: " (- ?no 1))
)
```

Dacă dorim putem rescrie regula *search* astfel încât să fie selectat un singur fiu şi de asemenea putem modifica prioritatea regulii add_to_level din valoarea implită 0 într-o valoare negativă (de exemplu –5) sau într-una pozitivă. Se poate introduce şi un arbore mai complet şi modificând strategia de aprindere a regulilor în agenda se pot urmări modificările obținute.

```
(deffacts noduri_arbore
   (nod A B C)  (nod B D E)  (nod C F G)  (nod D H I)  (nod E J K)  (nod F L M))
```

```
(defrule search
        (level ?no $? ?tata $?)
        (depth =(+ ?no 1))
        (nod ?tata $? ?fiu $?)
        =>
        (assert (adauga ?fiu))
        )
```

```
(defrule add_to_level
        (declare (salience –5))
        ?f1 <- (adauga ?fiu)
        ?f2 <- (level ?no $?val)
        (depth ?no)
        =>
        (retract ?f1 ?f2)
        (assert (level ?no $?val ?fiu)) )
```

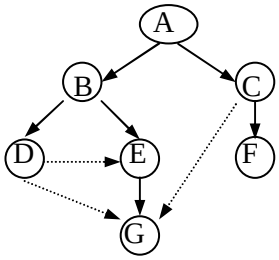
Strategy: *LEX, MEA* : *1-A* *2-B C* *3-D E F G* *4-H I J K L M*
Breadth, Depth, Simplicity: 1-A 2-C B 3-G D E F 4-I L M J K H

No salience for rule *add_to_level*

Strategy: LEX, MEA, Simplicity : 1-A 2-C B 3-G F D E 4-M L K J I H
Depth, : 1-A 2-B C 3-F G D E 4-J K H I L M
Breadth, Complexity : 1-A 2-C B 3-G D E F 4-I L M J K H

5.3. Verificarea unui arbore

Dat fiind faptul că un arbore este un caz particular de graf, diferențele între aceştia constau doar în datele de intrare. Parcurgerea unui arbore într-un sens pornind din orice nod, nu conduce la intrarea într-o buclă, fapt care se poate întâmpla într-un graf. Presupunând faptul că numele nodurilor sunt unice se poate verifica dacă un graf poate fi considerat a fi sau nu arbore. Cel mai simplu algoritm pentru a realiza acesta verificare este de lua parcurge graful pornind dintr-un nod oarecare şi în cazul în care nu se găsesc bucle înseamnă că acel graf poate fi considerat arbore. Pentru a ne folosi de avantajele CLIPS-ului vom realiza aceasta parcurgere pe toate drumurile posibile şi nu pe un singur drum.



Pas 1: (lista) (recent A) (new B C)
Pas 2: (lista A) (recent B C) (new D E F)
Pas 3: (lista A B C) (recent D E F) (new G)
S-a găsit o buclă...

După cum se poate observa, din exemplul de mai sus parcurgerea grafului se face pe nivele, mergând pe toate arcele o singură dată. Aceasta deplasare se aseamănă cu "valurile formate atunci când se aruncă o piatră într-un lac". Cheia implementării acestui algoritm constă în stocarea ultimelor noduri parcurse într-o listă *recent*. Pornind de la aceste noduri se vor verifica posibilele bucle realizate de nodurile imediat următoare. După cum se poate observa şi din figură există trei posibilități distincte de formare a unei bucle (marcate de săgețile punctate):

- două noduri din *recent* duc la acelaşi nod *new*. (caz 1)
- un nod *new* are legătură cu un nod mai "vechi" din *lista*. (caz 2)
- există un arc (legătură) între două noduri *new*. (caz 3)

```
(deffacts date_initiale
        (arc A B) (arc A C) (arc B D) (arc B E)  (arc C F) (arc E G) (arc D G)
        (arc B A) (arc C A) (arc D B) (arc E B)  (arc F C) (arc G E) (arc G D)
        (lista ) (recent) (adauga A))

(defrule nod_nou
        (lista $?old)
        (recent $? ?r $?)
        (arc ?r ?new&:(not (member$ ?new $?old)))       ;sa nu existe in lista
        =>
        (assert (new ?new))
)
```

```
(defrule gasit_bucla
        ?f<- (lista $?old)
        ?f1 <- (new ?n)
        (or (arc ?n ?o&:(member$ ?o $?old))                    ;caz 2
            ?f2 <- (new ?n &:(neq ?f1 ?f2))                    ;caz 1
            (and (new ?n2) (arc ?n ?n2))                       ;caz 3
        )
        =>
        (retract ?f) (printout t "Nu este arbore...")
)
(defrule adauga_nod
        (declare (salience -5))
        ?f1 <- (new ?n)
        ?f2 <- (adauga $?a)
        =>
        (retract ?f1 ?f2) (assert (adauga $?a ?n))
)
(defrule adauga->recent->old
        (declare (salience -20))
        ?f1 <- (old $?l)
        ?f2 <- (recent $?r)
        ?f3 <- (adauga $?a&:(neq 0 (length $?a)))
        =>
        (retract ?f1 ?f2 ?f3)
        (assert (old $?l $?r) (recent $?a) (adauga ))
)
```

Nodul din care se pleacă este A, dar acesta poate fi schimbat cu orice alt nod. Regula *găsit_bucla* poate fi rescrisă sub forma a trei reguli (fiecare regulă fiind asociată unuia din cele trei tipuri de buclă). Vom scrie numai pattern-urile celor trei reguli, deoarece RHS-ul acestora este identic:

```
?f1<-(new ?n)   ?f2 <- (new ?n &:(neq ?f1 ?f2))                ;caz 1
(lista $?old)   (new ?n)    (arc ?n ?o&:(member$ ?o $?old))    ;caz 2
(new ?n)        (new ?n2)   (arc ?n ?n2))                      ;caz 3
```

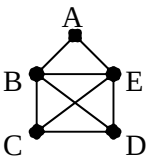
În primul caz se pune condiția ca cele două fapte identice *(new ?n)* să fie disticte, sau altfel spus să aibă adresele de fapt diferite *(neq ?f1 ?f2)*. După cum se ştie CLIPS-ul nu acceptă în mod implicit două fapte identice, şi din acest motiv trebuie dată la prompter comanda *(set-fact-duplication TRUE)*, sau modificată variabila de sistem din meniul *Execution*, submeniul *Options...* Dacă nu s-a găsit nici o buclă, elementele din *recent* sunt adăugate în *lista*, iar cele din *new* în *recent*. Deoarece graful studiat este bidirecțional, s-a preferat introducerea în lista de fapte a perechilor de fapte *(arc X Y) (arc Y X)*, în locul introducerii în fiecare regulă a

unui CE *or*, de fiecare dată când ar fi fost folosit un fapt *arc*. Dacă nu s-ar fi folosit astfel de perechi de fapte regula *găsit_buclă* ar fi arătat astfel:

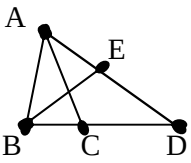
```
(defrule gasit_bucla
        ?f<- (lista $?old)
        ?f1 <- (new ?n)
        (or (or  (arc ?n ?o&:(member$ ?o $?old))                    ;caz 2
                 (arc ?o&:(member$ ?o $?old)  ?n)
            ?f2 <- (new ?n &:(neq ?f1 ?f2))                         ;caz 1
            (and  (new ?n2)  (or (arc ?n ?n2)
                                 (arc ?n ?n2))                      ;caz 3
        )
        =>
        (retract ?f) (printout t "Nu este arbore...") )
```

5.4. Parcurgerea unui graf trecând prin toate arcele o singură dată

Problema următoare a fost aleasă pentru a exemplifica un tip mai special de parcurgere a unui graf. Astfel se cere trasarea figurii următoare fără a trece peste o linie de două ori. Spre deosebire de problema comisului voiajor, unde se punea problema parcurgerii unui graf trecând prin toate nodurile o singură dată, în această problemă se cere parcurgerea grafului trecând prin toate arcele o singură dată.



.
1 A B
2 B C
3 C D
4 D E
5 E A
6 B E
7 B D
8 C E



.
1 A B
2 A C
3 A E
4 B C
5 B E
6 C D
7 E D

Datele de intrare pot fi citite dintr-un fişier text, în care până la introducerea caracterului "." se poate reprezenta grafic figura folosind caracterele ASCII. După aceasta se citesc pentru fiecare arc numărul asociat acestuia (acesta trebuie să fie unic) şi nodurile aflate la capetele sale.

Se doreşte afişarea tuturor soluțiilor posibile fără a restricționa programul să pornească cu un anume nod. Dacă se doreşte aceasta se şterge regula *punct-start* şi se introduce faptul *(drum X Y)* în construcția deffacts, unde X reprezintă numărul segmentului şi Y nodul din care se pleacă.

```
(deffacts date_initiale (nr_solutii 0))

(defrule citire_date
        =>
        (printout t "Fisier de intrare: ") (bind ?i (read))
        (if (eq (open ?i i) FALSE)
         then (printout t "Eroare la deschidere fisier.") (halt))
        (while (neq (bind ?line (readline i)) "." EOF)
                (printout t ?line t) )
        (while (neq (bind ?line (readline i)) EOF)
                (assert (segm (explode$ ?line))) )
        (close) )
```

```
(defrule punct_start
        (or  (segm ?n ?p ?)   (segm ?n ? ?p))       ; se iau toate segmentele
        =>                                          ; rând pe rând
        (assert (drum ?n ?p)) )

(defrule adauga_punct
        (drum $?f ?n1 ?p1)
        (or  (segm ?n2&~?n1 ?p1 ?p2)   (segm ?n2&~?n1 ?p2 ?p1))
        (test (not (member$  ?n2  $?f)))
        =>
        (assert (drum $?f ?n1 ?n2 ?p2)) )

(defrule solutie_finala
        (drum $?f ?)
        (forall   (segm ?n ? ?)
        (test (member$ ?n $?f)) )
        =>
        (assert (solutie $?f) (noduri)) )
```

În faptul *drum* sunt păstrate numerele tuturor segmentelor prin care s-a trecut, precum şi numele ultimului nod parcurs. La adăugarea unui nou nod la drumul parcurs, se asigură faptul că segmentul ales nu coincide cu ultimul arc parcurs şi că unul din capetele acestuia este identic cu ultimul nod prin care s-a trecut. Regula *soluție_finală* verifică dacă numerele tuturor arcelor sunt incluse în faptul *drum*. Pentru a simplifica timpul mare de calcul consumat de această regulă datorită CE *forall* ,verificat pentru fiecare fapt *drum* asertat, se poate introduce un fapt *(nr_arce ?n),* care poate fi citit direct din fişierul de intrare sau poate fi furnizat de o regulă după ce sunt numărate toate arcele. Acesta valoare poate fi stocată şi într-o variabilă globală. De exemplu:

```
(defglobal ?*n* = 0)
(defrule numara_segm (segm ? ? ?) => (bind ?*n* (+ ?*n* 1)) )
.............................
(defrule soluție_finală
        (drum $?f &:(eq ?*n* (length$ $?f)) ?)
        =>
        (assert (solutie $?f) (noduri)) )
```

Dacă prin folosirea variabile globale timpul nu scade deloc, prin folosirea unui fapt ordonat timpul în care sunt furnizate toate soluțiile scade la mai puțin de jumătate. Acest lucru demonstrează încă o dată că motorul de inferență lucrează mai bine cu faptele şi că programele trebuie să fie cât mai puțin procedurală. În bucla *while* din regula *citire_date* se vor număra arcele citite şi în final se va aserta un fapt *(nr_arce ?n),* de care se va folosi regula *soluție_finală.*

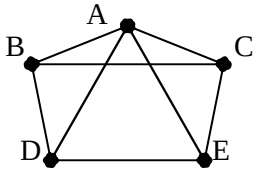
```
(defrule soluție_finală
        (nr_arce ?n)
        (drum $?f &:(eq ?n (length$ $?f)) ?last)
        =>
        (assert (solutie $?f ) (noduri ?last)) )
```

Regula *noduri_soluție* va forma lista *noduri* parcurgând faptul *drum* (ce conține numerele arcelor) în sens invers. Regula *afişare_soluții* realizează şi o contorizare a soluțiilor găsite, pentru ca în final în regula *start_again* să fie afişat şi numărul total al acestora.

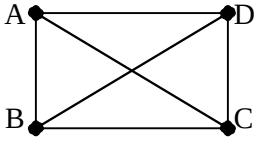
```
(defrule noduri_solutie
        (declare (salience 10))
        ?f1 <- (solutie $?f ?n)
        ?f2 <- (noduri ?p1 $?l )
        (or (segm ?n ?p1 ?p2) (segm ?n ?p2 ?p1))
        =>
        (retract ?f1 ?f2)
        (assert (solutie $?f) (noduri ?p2 ?p1 $?l)) )

(defrule afisare_solutie
        ?f1 <- (noduri $?sol)
        ?f2 <- (nr_solutii ?n)
        =>
        (retract ?f1 ?f2)
        (printout t "Solutie: " (expand$ $?sol) t)     ;(readline)
        (assert (nr_solutii (+ ?n 1))) )

(defrule start_again
        (declare (salience -10))
        (nr_solutii ?n)
        =>
        (printout t "Au fost gasite: " ?n " solutii." t t)
        (printout t "Doriti reluare (Y/N)?") (bind ?r (read))
        (if (neq ?r Y y YES Yes yes D d DA Da da)
         then (printout t "BYE !" t)
         else (reset) (run)) )
```

Dacă se doreşte încă o optimizare a problemei aceasta trebuie să țină cont de **nodurile pare** şi **nodurile impare** ale grafului. Prin definiție spunem că un nod este par sau impar în funcție de numărul segmentelor pe care le are drept capăt. După cum se poate observa un drum care parcurge toate arcele o singură dată, *porneşte şi se termină într-un nod impar* (dacă există astfel de noduri). Pentru prima figură reprezentată avem două noduri impare şi anume C şi D (fiecare având 3 segmente). Toate celelalte noduri sunt pare cu 2 şi respectiv 4 segmente. Dacă toate nodurile sunt pare, atunci se pot găsi soluții pornind din orice nod.

În concluzie problema nu are soluții dacă numărul "nodurilor impare" este diferit de 0 sau 2 (nodul de start şi de sfârşit). Vom implementa un set de reguli, care să stabilească paritatea / imparitatea unui nod şi care să stabilească dacă există soluții. Pentru exemplele următoare nu există soluții.



Noduri impare:
B, C, D, E



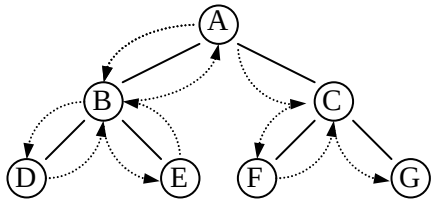
Toate cele 4 noduri sunt impare

```
(deffacts date_initiale (nr_solutii 0) (nod_impar ))

(defrule citire_date
        =>
        (printout t "Fisier de intrare: ") (bind ?i (read))
        (if (eq (open ?i i) FALSE) then (printout t "Fisier inexistent.") (halt))
        (while (neq (bind ?line (readline i)) "." EOF)  (printout t ?line t) )
        (set-fact-duplication TRUE)
        (bind ?na 0)
        (while (neq (bind ?n (read i)) EOF)
                (bind ?c1 (read i)) (bind ?c2 (read i)) (bind ?na (+ ?na 1))
                (assert  (nod ?c1)  (nod ?c2)  (segm ?n ?c1 ?c2)) )
        (assert (nr_arce ?na))
        (close) )

(defrule creare_nod
        (or (segm ?  ?c  ?) (segm ?  ?  ?c))
        (not (nod ?c ?))
        =>
        (assert (nod ?c 0)) )

(defrule paritate_nod
        ?f1 <- (nod ?c)
        ?f2 <- (nod ?c ?p)
         =>
        (retract ?f1 ?f2) (assert (nod ?c (+ ?p 1)) ) )

(defrule selectare_noduri_impare
        (declare (salince -1))
        (nod_impar $?n)
        ?f <- (nod  ?c  ?p&:(oddp ?p))
        =>
        (retract ?f) (assert (nod_impar $?n ?c)) )

(defrule no_solution
        (declare (salience -5))
        (nod_impar $?n &:(neq (length$ $?n) 0 2) )
        =>
        (printout t "Nu exista solutii" crlf) )

(defrule punct_start
        (declare (salience -5))
        (or (segm ?no ?p ?)
            (segm ?no ? ?p))
        (or (nod_impar $?n &:(eq (length$ $?n) 2)&:(member$ ?p $?n))
            (nod_impar $?n &:(eq (length$ $?n) 0)) )
        =>
        (assert (drum ?no ?p)) )
```

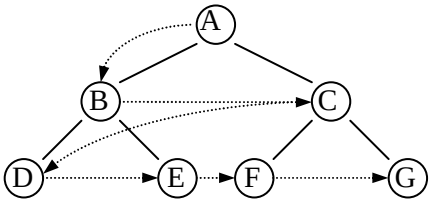
În regula punct_start se pune condiția de a începe dintr-un nod impar, doar dacă numărul acestor noduri este egal cu 2. Pentru cazul în care nu există noduri impare se pornește din orice punct. Celelalte reguli sunt identice.

5.5. Parcurgerea unui arbore în lățime şi în adâncime

Dat fiind un arbore vom prezenta asupra metodelor de parcurgere a acestora, folosindu-ne de facilitățile puse la dispoziție de un limbaj bazat pe reguli cum este CLIPS-ul. Pe lângă regulile realizate se vor studia şi efectele obținute prin modificarea strategiei de aprindere a regulilor în agendă. Pentru a oferi un caracter cât mai general acestor metode de parcurgere, arborii folosiți nu trebuie să fie neapărat binari, deci un nod părinte poate un număr oarecare de noduri fii.



Parcurgere în adâncime: A B D E C F G



Parcurgere în lățime: A B C D E F G

În figurile de mai sus parcurgerile arborilor au fost realizate de la stânga la dreapta. Dacă ar fi fost realizate în sens invers de la dreapta la stânga am fi obținut în adâncime A C G F B E D şi în lățime A C B G F E D.

Pentru toate exemplele următoare sunt definite o construcție deffacts cu ajutorul căreia se introduce arborele şi o regula ce caută rădăcina acestuia, deoarece de la acest nod se începe parcurgerea arborelui. Datele ar putea fi citite eventual dintr-un fişier, iar rădăcina ar putea fi introdusă din prima în faptele inițiale.

```
(deffacts noduri_arbore
        (nod A B C)  (nod B D E)  (nod C F G)  (nod E J K)  (nod F L M) )

 (defrule radacina_arbore
        (nod ?r $?)
        (not (exists (nod ? $? ?r $?)) )
        =>
        (assert (arbore ?r)) )
```

Vom defini o regulă simplă de parcurgere a unui arbore folosindu-ne de o listă (faptul ordonat *arbore*) în care ar trebui să se introducă noduri noi dacă nodul părinte este deja în listă şi nodul fiu încă nu a fost parcurs.

```
(defrule parcurgere_arbore
        ?a <- (arbore $?arb)
        (nod ?n $? ?f $?)
        (test (and (member$ ?n $?arb)  (not (member$ ?f $?arb))))
        =>
        (retract ?a) (assert (arbore $?arb ?f)) )
```

Afişare anormală parcurgere în lățime (*breadth*, *simplicity*, *compexity*)
Parcurgere în adâncime dr->stg (*MEA*, *LEX*)
Parcurgere în adâncime stg->dr (*depth*)

După cum se observă schimbând strategia motorului de inferență se pot obține pentru aceiaşi regulă mai multe tipuri de parcurgeri ale unui arbore. Singurul inconvenient pentru regula de mai sus îl constituie parcurgerea în lățime, pentru arborele definit în deffacts se obține în final faptul (arbore A C B E D G F K J M L). Deşi se observă o parcurgere pe nivele, ordinea nodurilor este dată de modul în care motorul de inferență găseşte fapte care se potrivesc cu pattern-ul regulii.

```
(defrule parcurgere_arbore
    ?a <- (arbore $?arb)
    (arbore $? ?n $?)
    (nod ?n  $?  ?f  $?)
    (test (not (member$ ?f  $?arb)))
    =>
    (retract ?a) (assert(arbore $?arb ?f )) )
```

Parcurgere în lățime (*depth*)
Parcurgere în adâncime dr->stg (celelalte)

```
(defrule parcurgere_arbore
    ?a <- (arbore $?arb)
    (arbore $? ?n $?)
    (or (nod ?n  $?  ?f  $?)
        (nod ?n  ?f  $?))
    (test (not (member$ ?f  $?arb)))
          =>
    (retract ?a) (assert(arbore $?arb ?f)) )
```

Parcurgere în lățime (*depth*)
Parcurgere în adâncime stg->dreapta

Deşi modificările făcute în cele două reguli par la prima inutile, se poate observa cum simpla introducere a pattern-ului *(arbore $? ?n $?)* schimbă ordinea în care motorul de inferență va găsi faptele ce vor activa regulile. În plus introducerea cu ajutorul unui CE *or* a două variante de selectare a fiului, va oferi o prioritate mai mare primului nod copil *(nod ?n ?f $?)*. În acest fel parcurgerea arborelui în adâncime va fi realizată acum în sens invers (de la stânga la dreapta).

Dacă se doreşte implementarea unor reguli care să realizeze numai un singur tip de parcurgere, atunci trebuiesc realizate mici modificări. Pentru o regulă care realizează doar deplasarea în adâncime, se oferă folosind un CE *or* două variante de selectare a unui nod din listă, prioritatea cea mai mare având-o alegerea ultimul nod parcurs (sau a ultimului element din faptul ordonat *arbore*).

```
(defrule depth
    ?a <- (arbore $?arb)
    (or (arbore $? ?n)
        (arbore $? ?n $?))
    (nod ?n $? ?f $?)
    (test (not (member$ ?f $?arb)))
    =>
    (retract ?a) (assert(arbore $?arb ?f )) )
```

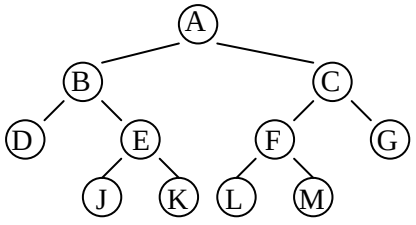
Parcurgere adâncime dr->stg

```
(defrule depth
    ?a <- (arbore $?arb)
    (or (arbore $? ?n)
        (arbore $? ?n $?))
    (or (nod ?n $? ?f $?)
        (nod ?n ?f $?))
    (test (not (member$ ?f $?arb)))
    =>
    (retract ?a) (assert(arbore $?arb ?f)) )
```

Parcurgere adâncime stg->dreapta

(diferit de *depth*)

Pas1: (arbore A) (nod A B *C*)
Pas2: (arbore A C) (nod C F *G*)
Pas3: (arbore A C G) (nod C *F* G)
Pas4: (arbore A C G F) (nod F L *M*)
Pas5: (arbore A C G F M) (nod F *L* M)
Pas6: (arbore A C G F M L) (nod A *B* C)
Pas7: (arbore A C G F M L B) .....



După cum se poate observa, regula *depth* va alege dacă se va putea mai întâi ultimul nod din listă şi apoi un nod aflat în interior. Unica problemă în faptul că pentru strategia *depth*, deşi se încearcă o parcurgere în adâncime aceasta nu este corectă. Lista obținută în acest caz este: (arbore A B D C F L E J G M K).

Pentru a realiza o parcurgere în lățime este suficient să introducem în listă pentru un nod părinte toți fii o dată. Se va observa însă faptul că rezultatele obținute sunt corecte doar pentru câteva strategii.

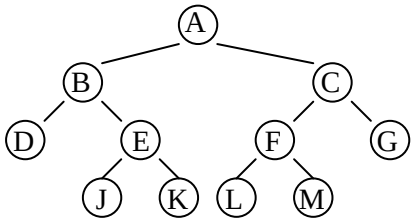
```
(defrule breadth
    ?f <- (arbore $?a)
    (nod ?n $?fii)
    (test (and (member$ ?n $?a)
               (not(subsetp $?fii $?a)) ))
    =>
    (retract ?f) (assert(arbore $?a ?fii)) )
```

Parcurgere corectă doar pentru strategiile (*breadth*, *simplicity*, *compexity*)

```
(defrule breadth
    ?f <- (arbore $?a)
    (arbore $? ?n $?)
    (nod ?n $?fii)
    (test (not (subsetp $?fii $?a)))
    =>
    (retract ?f) (assert(arbore $?a ?fii)) )
```

Parcurgere corectă doar pentru (*depth*)

Pas1: (arbore A)..... (nod A B C)
Pas2: (arbore A B C).....(nod B D E)
Pas3: (arbore A B C D E).....(nod C F G)
Pas4: (arbore A B C D E F G).....(nod E J K)
Pas5: (arbore A B C D E F G J K).... (nod F L M)
Pas6: (arbore A B C D E F G J K L M)



Pentru ultima regulă *breadth* doar pentru o singură strategie se obține o parcurgere corectă a arborelui (pentru celelalte strategii se obține ABCFGLMDEJK). Devine clar faptul că pentru a realiza o parcurgere corectă indiferent de strategia motorului de inferență este necesar controlul fiecărui nod ce va fi parcurs. Pentru aceasta ar trebui verificat pentru fiecare nod selectat, dacă nu există un alt nod plasat mai bine în lista.

Observația principală este aceea că pentru o parcurgere în lățime este avantajoasă alegerea unui nod aflat cât mai la stânga (cel mai aproape de începutul listei), iar pentru o parcurgere în adâncime cât mai spre dreapta.

```
(defrule next_nod
        ?f <- (arbore $?arb)
        (nod ?n1 $? ?f1 $?)
        (test (and (member$ ?n1 $?arb)  (not(member$ ?f1 $?arb)) ))
        (not (exists (nod ?n2 $? ?f2 $?)
                     (test (and (member$ ?n2 $?arb) (not(member$ ?f2 $?arb))
                                (< (member$ ?n2 $?arb) (member$ ?n1 $?arb)) )) ))
        =>
        (retract ?f) (assert (arbore $?arb ?f1))
)
```

Semnul '<' pentru parcurgere în lățime stg->dr (strategia *depth*); altfel dr->stg
Semnul '>' pentru parcurgere în adâncime stg->dr (strategia *depth*); altfel dr->stg

După cum se poate observa prin schimbarea semnului în testul de comparare a poziției celor două noduri *?n1* şi *?n2* (funcția *member$* returnează poziția unui element într-un multicâmp), se schimbă practic modul în care este parcurs arborele. Pentru a face independentă de strategie şi direcția de parcurgere, este necesară pe lângă selectarea nodului cât mai aproape de începutul sau sfârşitul listei şi selectarea fiului cât mai aproape de margine. Pentru o parcurgere de la stânga la dreapta trebuie în mod evident ales fiul cât mai din stânga.

```
(defrule next_nod
        ?f <- (arbore $?arb)
        (nod ?n1 $?fii)
        (nod ?n1 $? ?f1 $?)
        (test (and (member$ ?n1 $?arb)  (not(member$ ?f1 $?arb)) ))
        (not (exists (nod ?n1 $? ?f2 $?)
                (test (and (not(member$ ?f2 $?arb))
                        (> (member$ ?f2 $?fii) (member$ ?f1 $?fii)) )) ))
        (not (exists (nod ?n2 $? ?f3 $?)
                (test (and (member$ ?n2 $?arb) (not(member$ ?f3 $?arb))
                        (> (member$ ?n2 $?arb) (member$ ?n1 $?arb)) )) ))
        =>
        (retract ?f) (assert (arbore $?arb ?f1))
)
```

Dacă primul semn este '<' sensul de parcurgere este de la stânga la dreapta.
Dacă primul semn este '>' sensul de parcurgere este de la dreapta la stânga.
Dacă al doilea semn este '<' se realizează o parcurgere în lățime.
Dacă al doilea semn este '>' se realizează o parcurgere în adâncime.

Se vor exemplifica şi două variante mai apropiate de programarea procedurală. Va fi folosit un fapt *nr_elem* care va stoca numărul nodului curent. Acesta va fi incrementat sau decrementat în funcție de poziția la care se doreşte a se ajunge în listă. Acest fapt va fi asertat în regula *radacina_arbore* şi va avea inițial valoarea 1. După rularea programului acesta va conține numărul total al nodurilor din arbore incrementat cu unu.

```
(defrule parcurgere_in_latime
        ?f1 <- (arbore $?arb)
        ?f2 <- (nr_elem ?n)
        (nod =(nth$ ?n $?arb) $?fii)
        =>
        (retract ?f1 ?f2) (assert (arbore $?arb $?fii) (nr_elem (+ ?n 1)))
)

(defrule nu_exista_nod
        (arbore $?arb)
        ?f <- (nr_elem ?n &:(<= ?n (length $?arb)) )
        (not (exists (nod =(nth$ ?n $?arb) $?) ))
        =>
        (retract ?f) (assert (nr_elem (+ ?n 1)))
)
```

În regula *parcurgere_in_latime* este selectat din listă nodul părinte de pe poziția stocată de faptul *nr_elem*. Dacă acest nod nu există (poate fi nod frunză), atunci regula următoare *nu_exista_nod* va incrementa valoarea contorului pentru a parcurge tot arborele. Fără această regulă execuția programului s-ar fi terminat la întâlnirea primului nod terminal. Pentru a nu rula programul în buclă infinită, contorul este incrementat până când depăşeşte dimensiunea listei (au fost parcurse astfel toate nodurile).

Pentru parcurgerea în adâncime, dat fiind faptul că după întâlnirea unui nod terminal trebuie să ne întoarcem înapoi în listă cu una sau mai multe noduri, pentru a parcurge şi nodurile fiu rămase, variabila contor *nr_elem* va fi decrementată. Pentru aceasta a fost implementată regula *un_pas_inapoi*.

```
(defrule parcurgere_in_adancime
        ?f1 <- (arbore $?arb)
        ?f2 <- (nr_elem ?n)
        (nod =(nth$ ?n $?arb) $? ?fiu $?)
        (test (not (member$ ?fiu $?arb)))
        =>
        (retract ?f1 ?f2)
        (assert (arbore $?arb ?fiu) (nr_elem (+ (length $?arb) 1)) )
)

(defrule un_pas_inapoi
        (declare (salience -10))
        ?f <- (nr_elem ?n &:(> ?n 1))
        =>
        (retract ?f) (assert (nr_elem (- ?n 1)))
)
```

Pentru strategia *depth* parcurgere arbore stg->dr
Pentru celelalte strategii parcurgere arbore dr->stg

Spre deosebire de cazul anterior, pentru parcurgerea în adâncime dacă există în listă nodul de pe poziția *nr_elem* se trece direct la ultimul nod introdus în faptul ordonat *arbore* (valoarea contorului nu mai este doar incrementată). Dacă valoarea contorului este mai mică decât unu, înseamnă că s-a ajuns înapoi în rădăcină şi că nu mai există nici un nod care nu a fost parcurs.

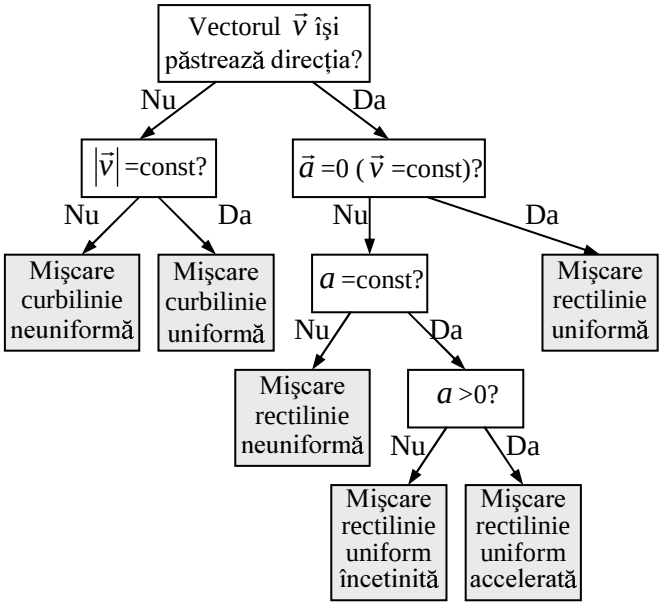
5.6. Implementarea unui arbore de decizie

Arborii de decizie sunt utilizați în special pentru rezolvarea unor probleme de clasificare. Pentru început trebuie ales pentru fiecare întrebare un răspuns dintr-un set de răspunsuri valide. În funcție de aceste răspunsuri sunt eliminate o serie de soluții din setul actual de soluții posibile ale problemei. Cu cât se coboară mai mult în arbore, cu atât acest set de soluții posibile se micşorează. În final soluția furnizată corespunde nodului frunză în care s-a ajuns

Un arbore de decizie este format din noduri şi arce. Nodurile reprezintă locațiile arborelui şi pot fi noduri de decizie sau noduri soluție. Arcele reprezintă

conexiunile între nodurile părinte şi nodurile copil şi numărul acestora este legat de setul de răspunsuri valide pe care le poate furniza utilizatorul pentru o întrebare. Deşi deplasarea în arbore se face de sus în jos, există şi situaţii când poate fi necesară şi deplasarea de jos în sus (de exemplu într-un arbore care învaţă, atunci când setul de întrebări trebuie modificat, deoarece au fost găsite soluţii noi sau vechile întrebări nu sunt bine formulate).

Cei mai utilizaţi arbori de decizie sunt cei binari, în care pentru fiecare nod întrebare sunt posibile doar două răspunsuri (de obicei de tip afirmativ sau negativ "Yes/No"). În exemplul de mai jos se realizează clasificarea mişcărilor unui punct material. Sunt luate în consideraţie doar cele mai importante caracteristici ale mişcării vectorul viteză şi vectorul acceleraţie. În funcţie de valorile acestora se realizează o clasificare primară a mişcării. Deşi arborele de decizie reprezentat mai jos nu este foarte mare (are caracter didactic), odată cu introducerea altor parametri dimensiunile acestuia pot creşte foarte mult.



În exemplele CLIPS-ului sunt furnizate două exemple unul de clasificare a animalelor şi altul pentru alegerea unui vin cât mai indicat pentru un meniu. Au fost implementate şi probleme de diagnoză, în care se încerca găsirea unui remediu pentru o maladie, dintr-un set de remedii posibile prin identificarea cauzelor şi implicit a bolii de care suferea pacientul.

Baza de cunoştinţe poate fi formată din fapte compuse (deftemplate) sau fapte simple. Programatorul alege în funcţie de complexitatea problemei şi implicit a arborelui de decizie rezultat, modul în care va fi organizată baza de date. Vor fi prezentate pentru aceiaşi problemă două variante de program , prima utilizând fapte compuse, iar a doua fapte ordonate. Prin simpla schimbare a arborelui de decizie, problema poate fi aplicată foarte uşor şi în alte domenii.

În varianta următoare nu va fi folosită nici o instrucțiune procedurală (*if*, *while*, *loop-for-count*). De exemplu bucla în care se cere reintroducea unui răspuns invalid (diferit de "da" sau "nu"), este realizată de regula *răspuns_greşit* în care este şters faptul, şi de regula reactivată de patern-ul *(not (raspuns ?))*.

```
(deftemplate nod
    (slot nume) (slot tip) (slot intrebare) (slot nod-da) (slot nod-nu) (slot raspuns))

(defrule initializare
        (not (nod (nume root)))
        =>
        (load-facts "date.dat")  (assert (nod-curent root)) )

(defrule intrebare-nod-de-decizie
        (nod-curent ?nume)
        (nod (nume ?nume) (tip decizie) (intrebare ?intrebare))
        (not (raspuns ?))
        =>
        (printout t ?intrebare " (da/nu) ")  (assert (raspuns (read))) )

(defrule raspuns-gresit
        ?f <- (raspuns ~da&~nu)
        =>
        (retract ?f) )

(defrule ramura-da
        ?f1 <- (nod-curent ?nume)
        (nod (nume ?nume) (tip decizie) (nod-da ?ramura))
        ?f2 <- (raspuns da)
        =>
        (retract ?f1 ?f2)  (assert (nod-curent ?ramura)) )

(defrule ramura-nu
        ?f1 <- (nod-curent ?nume)
        (nod (nume ?nume) (tip decizie) (nod-nu ?ramura))
        ?f2 <- (raspuns nu)
        =>
        (retract ?f1 ?f2)  (assert (nod-curent ?ramura)) )

(defrule afisare-nod-raspuns
        (nod-curent ?nume)
        (nod (nume ?nume) (tip raspuns) (raspuns ?valoare))
        (not (raspuns ?))
        =>
        (printout t "Cred ca este vorba de " ?valoare crlf)
        (printout t "Corect ?(da/nu)")  (assert (raspuns (read))) )

(defrule raspunsul-este-bun
        ?f1 <- (nod-curent ?nume)
        (nod (nume ?nume) (tip raspuns))
        ?f2 <- (raspuns da)
        =>
        (retract ?f1 ?f2) (assert (continuare)) )
```

```
(defrule raspunsul-nu-este-bun
        ?f1 <- (nod-curent ?nume)
        (nod (nume ?nume) (tip raspuns))
        ?f2 <- (raspuns nu)
        =>
        (retract ?f1 ?f2)  (assert (inlocuieste-nodul ?nume)) )

(defrule inlocuieste-raspuns-gresit
        ?f1 <- (inlocuieste-nodul ?nume)
        ?f2 <- (nod (nume ?nume) (tip raspuns) (raspuns ?valoare))
        =>
        (retract ?f1)
        (printout t "Care este noua soluție?")  (bind ?solutie-noua (read))
        (printout t "Intrebarea care diferentiaza " ?solutie-noua " de " ?valoare "? ")
                (bind ?intrebare (readline))
        (printout t "De acum voi sti că răspunsul corect este  " ?solutie-noua crlf)
        (bind ?nodnou1 (gensym*))
        (bind ?nodnou2 (gensym*))
        (modify ?f2 (tip decizie) (intrebare ?intrebare)
                        (nod-da ?nodnou1) (nod-nu ?nodnou2) (raspuns nil))
        (assert (nod (nume ?nodnou1) (tip raspuns) (raspuns ?solutie-noua)))
        (assert (nod (nume ?nodnou2) (tip raspuns) (raspuns ?valoare)))
        (assert (continuare)) )

(defrule continuare
        (continuare)
        (not (raspuns ?))
        =>
        (printout t "Mai incercam? (da/nu) ")  (assert (raspuns (read))) )

(defrule reluam
        ?f1 <- (continuare)
        ?f2 <- (raspuns da)
        =>
        (retract ?f1 ?f2) (assert (nod-curent root)))

(defrule stop
        ?f1 <- (continuare)
        ?f2 <- (raspuns nu)
        =>
        (retract ?f1 ?f2) (save-facts "date.dat" local nod) )
```

Se observă faptul că arborele de decizie poate învăța. Dacă utilizatorul consideră că nodul frunză în care a ajuns nu este soluția cea mai potrivită (existând incertitudini datorită unei alte soluții), atunci acesta va formula o altă întrebare şi va preciza noua soluție (aceasta corespunde răspunsului afirmativ la întrebarea formulată). Datele de intrare sunt citite cu ajutorul comenzii *load-facts* şi salvate în final cu *save-facts*. Se salvează practic numai faptele deftemplate nod locale (vizibile numai în modulul curent MAIN). Funcțiile **gensym*** sunt folosite pentru a genera un simbol unic. În acest fel se evită existența a două noduri care au acelaşi

nume. Dacă se dorește reluarea programului, atunci este suficient ca rădăcina arborelui de decizie să devină *nod-curent*. Pentru arborele de decizie reprezentat mai sus faptele din fişierul *"date.dat"* pot fi următoarele:

```
(nod (nume  root) (tip decizie) (întrebare "Vectorul v îşi păstrează întotdeauna direcția?")
                (nod-da  n1) (nod-nu  n2))
(nod (nume  n1) (tip decizie) (întrebare "Viteza este constantă?") (nod-da  n3) (nod-nu  n4))
(nod (nume  n2) (tip decizie) (întrebare "Accelerația este nulă?") (nod-da  n5) (nod-nu  n6))
(nod (nume  n3) (tip raspuns) (raspuns "Mişcare curbilinie neuniformă"))
(nod (nume  n4) (tip raspuns) (raspuns "Mişcare curbilinie uniformă"))
```

După cum se poate observa în construcția deftemplate **nod** există sloturi care nu sunt folosite decât de nodurile decizie (cum ar fi *intrebare, nod-da, nod-nu*) şi un slot *raspuns* utilizat doar de nodurile soluție. Singurele sloturi folosite de toate nodurile în comun sunt *nume* şi *tip*.

În varianta următoare vor fi folosite fapte ordonate în locul faptelor deftemplate nod. Un nod va arăta de această dată astfel:

```
(nod  n1  "Viteza este constantă?" n3  n4)    (nod n4 "Mişcare curbilinie uniformă")
```

De această dată diferențele între un nod de tip decizie şi cele de tip soluție constau doar în nodurile copil enumerate la finalul faptului, pe care un nod terminal nu are. Devine foarte uşor de implementat un arbore de decizie care nu este binar (poate avea mai mult de două noduri fiu) prin simpla introducere a acestora în listă.

```
(nod  tata  "Ce înălțime are persoana (1-mică, 2-medie, 3-mare, 4-f.mare)?" f1 f2 f3 f4)
```

Pentru selectarea nodului copil ales se foloseşte *(nth$ (read) $?fii)*, unde valoarea introdusă de la tastatură trebuie să fie cuprinsă între 1 şi *(length$ $?fii)*. Faptele citite din fişierul *"date.dat"* trebuie sa fie bineînțeles fapte ordonate.

```
(defrule initializare
        (not (nod $?))
        =>
        (load-facts "date.dat") (assert (go_to root)) )

(defrule select_nod
        ?f <- (go_to ?name)
        (nod ?name  ?string  $?fii)
        =>
        (printout  t  t ?string)
        (if (neq (length$ $?fii) 0)                    ;daca nu este nod terminal
         then
                (retract ?f)  (printout  t  " (da/nu) ")
                (while (neq (bind ?r (read)) Da da DA d D Nu nu NU n N) do )
                (if (neq ?r Da da DA d D)
                 then (assert (go_to  (nth$ 1 $?fii)))
                 else (assert (go_to  (nth$ 2 $?fii))) )
)       )
```

Deşi putem modifica regulile folosite anterior pentru a implementa un arbore de decizie care învață, vom realiza o regulă nouă mult mai compactă, care le înglobează pe toate la un loc. Oricum programul rulează şi fără regula următoare.

```
(defrule nod_terminal_nou
    ?f1 <- (go_to ?name)
    ?f2 <- (nod ?name  ?valoare)
    =>
    (retract ?f1) (printout  t  t "Corect ?(da/nu) ")
    (if (neq (read) Da da DA d D)
     then
         (printout t "Care este noua soluție?")  (bind ?solutie-noua (readline))
         (printout t "Intrebarea care diferentiaza " ?solutie-noua " de " ?valoare "? ")
         (bind ?intrebare (readline)) (retract ?f2)
         (assert  (nod  (bind ?nodnou1 (gensym*))  ?solutie-noua)
                  (nod  (bind ?nodnou2 (gensym*)) ?valoare)
                  (nod  ?name  ?intrebare  ?nodnou2  ?nodnou1) )
    )
    (printout  t "Mai incercam? (da/nu) ")
    (if (neq (read) Da da DA d D)
     then  (save-facts "date.dat")
     else (assert (go_to  root)) )
)
```

5.7. Sistem Expert pentru diagnosticarea unei imprimante

Dacă în exemplul anterior, toate nodurile arborelui de decizie erau reprezentate prin întrebări, excepție făcând nodurile terminale (sau frunzele) care reprezentau soluțiile SE, de această dată în afara faptului că nu se mai respectă structura arborescentă pot fi întâlnite în interiorul grafului şi indicații (instrucțiuni) în afara întrebărilor. Trecerea de la structura arborescentă la o structură de graf este foarte simplă de realizat, datorită flexibilității cu care a fost proiectat SE. Programatorul trebuie practic să se asigure de unicitatea etichetelor asociate nodurilor şi de corectitudinea acestora (acestea trebuie să respecte schița pe care a proiectat-o în prealabil).

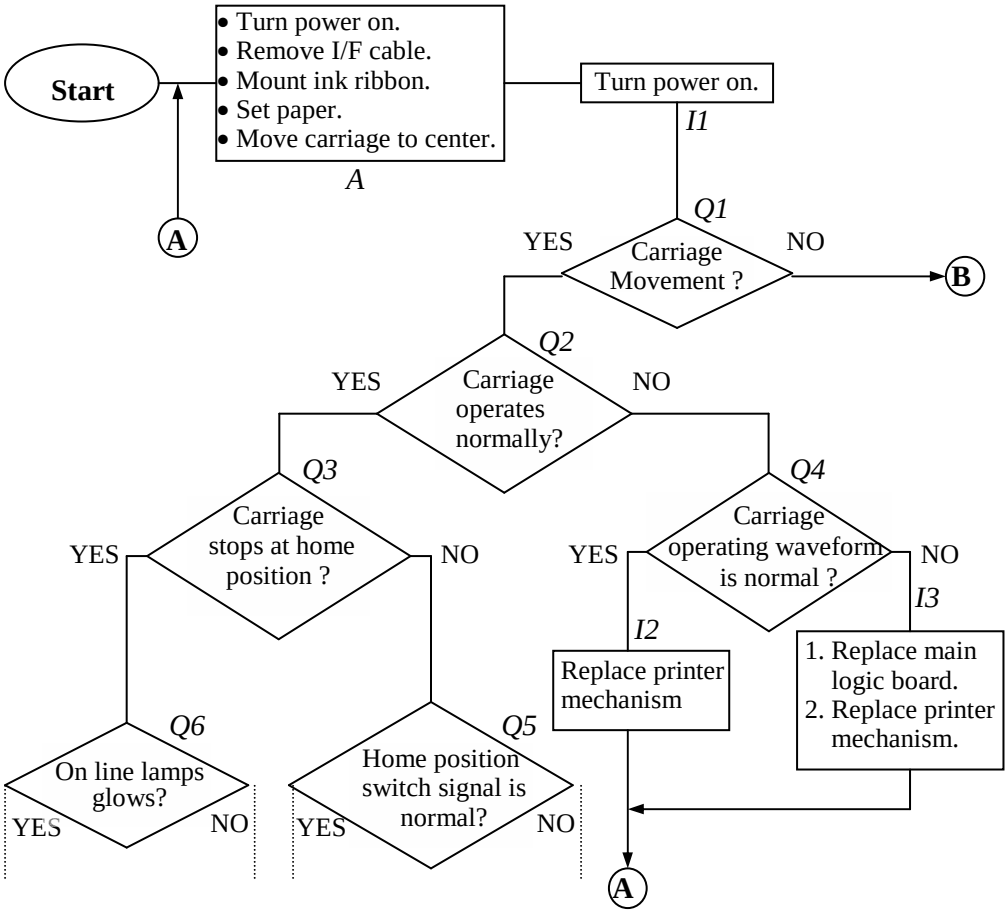
Au fost realizate două reguli, una pentru a prelua nodurile de tip întrebare şi cealaltă pentru nodurile de tip instrucțiune.

```
(defrule select_question
        ?f <- (go ?g)
        (instr ?g ?yes ?no ?i)
        (test (str-index "?" ?i))
        =>
        (printout t t ?i "(Yes/No) ")
        (while (neq (bind ?r (read)) Yes Y y yes YES No N n no NO)
                (printout t "Valid answer (Yes/No) :") )
        (retract ?f)
        (if (neq ?r No N n no NO)
         then   (assert (go ?yes))
         else    (assert (go ?no)) )
)
```

```
(defrule select_instruction
        ?f <- (go ?g)
        (instr ?g ?next $?instr)
        (test (not (str-index "?" (implode$ $?instr))))
        =>
        (loop-for-count (?i 1 (length$ $?instr))
                (printout t (nth$ ?i $?instr) crlf)
        )
        (retract ?f)  (readline)  (assert (go ?next)) )
```

În continuare va fi prezentată doar o mică parte din schema bloc elaborată pentru depanarea defecțiunilor unei imprimante. Utilizatorul sistemului expert este îndrumat pas cu pas, astfel încât la sfârşitul parcurgerii acestuia este găsită defecțiunea, eventual chiar şi eliminată.

Poate fi observată foarte clar prezența nodurilor de tip instrucțiune în fața şi după nodurile de tip întrebare, precum şi existența unor puncte de salt ce conduc la formarea unor bucle în care sunt testare diferite părți componente ale imprimantei.



Datele de intrare pot fi citite dintr-un fişier sau pot fi introdu-se într-o comandă deffacts. Faptul *go* conține locația curentă în arborele de decizie şi are la început valoarea *A*. Programul prezintă marele avantaj că se pot asocia pentru noduri orice nume (simboluri, şiruri de caractere şi chiar numere).

```
(deffacts date-initiale
        (go A)
        (instr 1  2  "Turn power off." "Remote I/F cable." "Mount ink ribbon."
                "Set paper." "Move carriage to center.")
        (instr I1 Q1 "Power on.")
        (instr Q1 Q2 B "Carriage movement?")
        (instr Q2 Q3 Q4 "Carriage operates normally?")
        (instr Q4 I2 I3 "Carriage motor operating wave form is normal ? *1")
        (instr I2 A "Raplace printer mechanism.")
        (instr I3 A "1) Repace min logic board.2) Repace printer mechanism.")
        (instr Q3 Q6 Q5 "Carriage stops at home position?")
.....
```

Regulile *select_question* şi *select_instruction* deosebesc nodurile decizie de nodurile soluție in funcție de prezența sau absența "semnului întrebării". Fără introducerea acestui semn de punctuație în mesajul întrebărilor programul nu funcționează corect.

5.8. Program Expert de diagnosticare a defecțiunilor unui calculator

O altă metodă de implementare a unui arbore de decizie, constă în realizarea unei reguli pentru fiecare nod părinte. În acest fel numărul regulilor creşte odată cu dimensiunea arborelui de decizie. Avantajele pe care le prezintă această metodă constau în folosirea unui număr foarte mic de fapte, ceea ce duce la o viteză foarte mare a motorului de inferență (procesul de potrivire "matching" se realizează foarte rapid). Ținând cont însă de faptul că pentru fiecare nod de tip întrebare se aşteaptă introducerea unui răspuns de la tastatură, se poate afirma faptul că viteza nu este factorul principal de care trebuie să se țină cont pentru elaborarea unui arbore de decizie (acesta constă în modul în care întrebările formulate acoperă domeniul soluțiilor).

Pentru a verifica validitatea răspunsurilor introduse au fost definite funcțiile *întreb* şi *da-sau-nu*, în prima se verifică dacă răspunsul oferit de utilizator este valid, altfel spus dacă este simbol sau şir de caractere (de tip *lexeme*) şi dacă face parte din lista răspunsurilor admise. Această listă este furnizată de cea de-a doua regulă *da-sau-nu*, care returnează simbolul TRUE sau FALSE în funcție de caracterul afirmativ sau negativ al răspunsului. Aceste simboluri vor duce la selectarea părții then sau a părții else, (corespunzătoare celor doi fii ai nodului părinte) din instrucțiunea procedurală *if* a regulii activate. Se poate afirma faptul că regula *determinare_stare_de_functionare* corespunde nodului rădăcină a arborelui, fiind întotdeauna prima activată. Dacă s-a găsit o soluție (nod frunză), cu ajutorul regulii *afisare_reparatie* comenzilor **(reset) (retract *)** sunt şterse toate faptele şi execuția programului se opreşte (altfel "trebuie mers la service cu computerul").

```
(deffunction intreb (?intrebare  $?v)                    ;$?v - valori permise
    (printout t  ?intrebare)
    (while (or (not(lexemep(bind ?r (read)))) (not(member$(lowcase ?r) $?v)) )
        (printout  t  "Raspunsuri valide " $?v ":") )
    (lowcase ?r) )

(deffunction da-sau-nu (?intrebare)
        (bind ?rsp (intreb ?intrebare da nu d n))
        (if (or (eq ?rsp da) (eq ?rsp d))   then TRUE    else FALSE))
-------------------------------------------------------------------------------------------------
(defrule determinare_stare_de_functionare
        (not (stare_de_functionare computer ?))
        (not (repair ?))
        =>
        (if (da-sau-nu "Computerul porneste (da/nu)? ")
         then  (if (da-sau-nu "Calculatorul functioneaza normal ? (da/nu)")
                 then (assert(repair "Computerul Dvs. nu necesita nici o reparatie !!!"))
                 else (assert (stare_de_functionare computer ne_coresp)) )
        else    (assert (stare_de_functionare computer nu_porneste)) ))

(defrule fara_reparatie
        (declare (salience -10))
        (not (repair ?))
        =>
        (assert (repair "Va trebui sa duceti computerul la service !!!")))

(defrule afisare_reparatie
        (declare (salience 10))
        (repair ?item)
        =>
        (printout t t t "Reparatie sugerata :" ?item t)  (reset) (retract *))
-------------------------------------------------------------------------------------------------
(defrule zgomot_ventilator
        (stare_de_functionare computer nu_porneste)
        =>
        (if (da-sau-nu "Functioneaza ventilatorul sursei ?(da/nu) ")
         then  (assert (ventilator computer se_roteste))
         else  (assert (ventilator computer nu_se_roteste)) ))

(defrule verificare_alimentare
        (ventilator computer nu_se_roteste)
        =>
        (if (da-sau-nu "Computerul este alimentat ?(da/nu) ")
         then  (assert (sursa computer alimentata))
         else  (assert (repair "Alimentati computerul !")) ))

(defrule verificare_cablu_alimentare
        (sursa computer alimentata)
          =>
        (if (da-sau-nu "Cablul de alimentare este bun ? (da/nu) ")
         then  (assert (repair "Schimbati sursa de alimentare a computerului  !"))
         else  (assert (repair "Schimbati cablul de alimentare !")) ))
```

```
(defrule verificare_CPU
        (or  (ventilator computer se_roteste)
             (hdd computer nu_functioneaza) )
        =>
        (if (da-sau-nu "Procesorul functioneaza ? (da/nu) ")
         then  (assert (cpu computer functioneaza))
         else   (assert (repair "Schimbati Procesorul computerului  !"))        ))

(defrule verificare_memorie
        (cpu computer functioneaza)
        =>
        (if (da-sau-nu "Memoria RAM functioneaza ? (da/nu) ")
         then  (assert (memorie computer functioneaza))
         else   (assert (repair "Schimbati Memoria RAM a computerului  !")) ))

(defrule verificare_placa_video
        (memorie computer functioneaza)
          =>
        (if (da-sau-nu "Placa video functioneaza ? (da/nu) ")
         then  (assert (placa_video computer functioneaza))
         else   (assert (repair "Schimbati Placa video a computerului  !")) ))

(defrule verificare_placa_baza
        (placa_video computer functioneaza)
        =>
        (if (da-sau-nu "Placa de baza functioneaza ? (da/nu) ")
         then  (assert (memorie computer functioneaza))
         else   (assert (repair "Schimbati Placa de baza a computerului  !")) ))

(defrule start_error
        (stare_de_functionare computer ne_coresp)
        =>
        (if (da-sau-nu "Se semnaleaza vreo eroare la initializare ? (da/nu) ")
         then  (assert (start computer error))
         else   (assert (start computer no_error)) ))

(defrule diagnostic_mem
        (mem computer error)
        =>
        (assert (repair "Vedeti daca memoria a fost corect instalata pe placa de baza.
                                Daca da, mergeti cu placa de baza la service.")) )
(defrule mem_error
        (start computer error)
        =>
        (if (da-sau-nu "Se semnaleaza eroare la numararea memoriei ? (da/nu) ")
         then  (assert (mem computer error))
         else   (assert (mem computer no_error)) ))

(defrule bios_error
        (mem computer no_error)
        =>
        (if (da-sau-nu "Se semnaleaza eroare de CHECKSUM la BIOS ? (da/nu) ")
```

```
        then   (assert(repair "Intrati in Bios si dati LOAD BIOS DEFAULT si verificati
                              bateria de pe placa de baza !!!"))
        else   (assert (bios computer no_error))    ))

(defrule startHDD_error
        (bios computer no_error)
        =>
        (if (da-sau-nu "Se semnaleaza eroare la initializarea HDD-ului ? (da/nu) ")
         then  (assert (repair "Vedeti daca HDD-ul a fost corect instalat.
                               Daca nu, mergeti cu HDD-ul la service !")) ))
(defrule SO_error
        (start computer no_error)
        =>
        (if (da-sau-nu "Se semnaleaza vreo eroare la incarcarea SO ? (da/nu) ")
         then  (assert (repair "Incercati sa reinstalari Sist.Op. sau sa reconfigurati
                    device-urile prezente in computer (citind documentatia lor ) !"))
        else   (assert (SO computer no_error)) ))

(defrule SO_se_blocheaza
        (SO computer no_error)
        =>
        (if (da-sau-nu "Se blocheaza Sistemul de Oprerare ? (da/nu) ")
         then  (assert (SO_b computer error))
         else  (assert (SO_b computer no_error)) ))

(defrule cooler_test
        (SO_b computer error)
        =>
        (if (da-sau-nu "Coolerul-ul functioneaza ? (da/nu) ")
         then  (assert (cooler computer nu_functioneaza))
         else  (assert (repair "Schimbati cooler-ul !"))      ))

(defrule hdd_test
     (cooler computer nu_functioneaza)
     =>
     (if (da-sau-nu "Hdd-ul functioneaza ? (da/nu) ")
     then      (assert (hdd computer nu_functioneaza))
     else (assert(repair "Incercati sa reparati erorile de pe Hdd sau sa-l schimbati!")) ))

(defrule device_error
        (SO_b computer no_error)
        =>
        (if (da-sau-nu "Sunt device-uri care nu functioneaza ? (da/nu) ")
         then  (assert (device computer error))
         else  (assert (repair "Computerul Dvs. nu necesita nici o reparatie !!!")) ))

 (defrule device_solve
        (device computer error)
        =>
        (if (da-sau-nu "Ati reinstalat acele device-uri care nu functioneaza ?(da/nu) ")
         then else  (assert (repair "Incercati sa instalati(reinstalati) device-urile care
                                    nu functioneaza corespunzator !")) ))
```

5.9. Sistem Expert în Lingvistică

Următorul program primeşte la intrare un cuvânt şi la ieşire va furniza acel cuvânt despărţit în silabe. Pentru a implementa un astfel de SE vom construi un set de reguli, care să identifice grupările de consoane şi vocale dintr-un cuvânt şi să găsească metoda cea mai indicată pentru formarea de noi silabe.

Pentru aceasta ne vom folosi, pentru început de regulile furnizate de *"Îndreptarul ortografic, ortoepic şi de punctuaţie"*, elaborat de Institutul de Lingvistică al Universităţii din Bucureşti. şi de *"Îndreptarul de sintaxă, fonetică, ortografie, ortoepie şi punctuaţie"* a cărui autori sunt Gh. Badea şi Const. Miu.

1. Regula principală de formare a unei silabe este dedusă din însăşi definiţia acesteia:

"Silaba este unitatea sonoră alcătuită dintr-o vocală sau dintr-un grup de sunete, *care cuprinde în mod obligatoriu o vocală* şi care se pronunţă printr-o singură deschidere a gurii".

2. Când între două vocale se află o consoană, aceasta trece în cea de-a doua silabă. Vom nota acest caz cu **V-CV** (sunt incluse automat cazurile V-CD, D-CV, D-CD, D-CT, T-CV, T-CD, unde notaţiile V-vocală, C-consoană, D-diftong şi T-triftong). Exemple: *o-ră, a-tom, ma-rea, dea-să, floa-rea,* bo-*ie-roai*-că, le-*oai-că,* zo-*ioa-sei*.

3. Dacă vocala este urmată de două sau mai multe consoane, prima consoană trece în silaba dinainte şi cealaltă (celelalte) în silaba următoare. VC-CV, VC-CCV... Exemple: *ban-că, as-pru, con-tra, mon-stru, ab-strac*-ţi-e.

4. Fac excepţie de la regula de mai sus cazurile în care:

- prima consoană este b, c, d, f, g ,h ,p ,t ,v, iar a doua este l sau r, amândouă consoanele trec la silaba următoare V-CCV. Ex: *ci-fră, sti-clă, a-cru, du-hli*.
- grupurile ct, cţ, şi pt, pţ precedate de consoane se despart. Deoarece pe lângă aceste grupuri au fost găsite altele putem reformula aceasta regulă. Pentru grupurile de consoane l**pt**, mpt, m**pţ**, n**cs**, n**ct**, n**cţ**, n**cv**, n**dv**, rct, r**tf**, s**tm** despărţirea în silabe se face astfel VCC-CV. Exemple: *sculp-tor, somp-tu*-os, *sfinc-şii, punc-taj, func-ţi*-e,de-*linc-vent, sand-viş,* an-*tarc-tic, jert-fă, ast-mă.*
- grupurile de litere che, chi, ghe, ghi redau o singură consoană redau o singură consoană şi despărţirea în silabe se realizează ca şi în cazul V-CV.
- sunt întâlnite în neologisme grupări consonantice neobişnuite şi despărţirea se face după consoana a doua V**CC**-C...V (Ex: ang-strom, horn-blendă, tung-sten). Exemple: *e-cher*, u-*re-chi*, *ve-ghe, ne-ghi*-nă.

De asemenea uneori pot fi întâlnite şi situaţii speciale în care regula VC-CV este respectate. De exemplu dacă:

- se întâlneşte litera *x*, care notează două consoane cs şi gz , urmată sau nu de alte consoane (*a-xă, e-xa*-men, *tex-til, ex-plo*-zi-e, *dex-te*-ri-ta-te).
- când se întâlnesc două consoane duble (*în-no*-bi-la, in-*ter-regn*, for-*tis-si*-mo).

- prima consoană este l sau r, iar a doua este b, c, d, f, g ,h ,p ,t ,v (situație inversă excepției de mai sus) sau prima consoană l şi a doua r.

5. Când două vocale succesive nu formează un diftong, sau altfel spus se află în hiat, fiecare face parte din altă silabă **V-V**. În caz contrar cele două vocale ar fi fost incluse în aceiaşi silabă. (*a-er, a-ici,* na-*ți-u*-ne, *po-et, ca-is,* con-ti-*nu-a*).

6. Când o vocală este urmată de un diftong sau triftong **V-D**, **V-T**, vocala face parte din prima silabă (*ba-ia,* fe-*me-ie, no-uă, le-oai*-că, *ro-iau, su-iai*). Se poate afirma şi faptul că semivocalele *i* şi *u* între două vocale trec la silaba următoare.

7. Vocala *i* este uneori nesilabică şi se mai numeşte i *scurt* (*şoptit, afonizat*). Este cazul în care acesta apare singură la sfârşitul cuvântului, neînsoțită de altă vocală, (în special pentru formele de plural, şi persoana II indicativ prezent la verbelor). Exemple: bun*i*, cărț*i*, vez*i*, pom*i*, vez*i*, lu-crez*i*, dați.

Nu vor fi studiate şi cazurile în care două cuvinte întregi sunt rostite împreună şi trebuie unite prin *cratimă*. De asemenea nici situațiile în care se pentru cuvintele compuse şi cele derivate cu prefixe (şi la unele derivate cu sufixe, respectiv cele care se termină în grupuri consonantice şi sufixul începe cu o consoană) despărțirea în silabe se face , de preferință ținându-se cont de părțile componente: *de-spre* (nu des-pre), *in-egal* (nu i-ne-gal), *ne-stabil* (nu nes-tabil), *sub-linia* (nu su-blinia), *drept-unghi* (nu drep-tunghi), *vârst-nic* (nu vârst-nic).

```
(deffunction vocala (?v) (member$ ?v  (create$ a e i o u)) )

(deffunction consoana (?c) (str-index ?c “b c d f g h j k l m n p q r s t v w x y z”)) )

(deffunction diftong (?v1 ?v2) (str-index (str-cat ?v1 ?v2)
                                  "ea eo ia ie io iu oa ua ue ai ii ei oi ui au eu iu ou uu"))

(deffunction triftong (?v1 ?v2 ?v3)
    (member$ (sym-cat ?v1 ?v2 ?v3) (create$ eoa ioa eai eau iai iei iau oai uau)) )

(defrule introducere_cuvant
        =>
        (printout t "Introduceti un cuvânt de la tastatură: " t) (assert (word (read))) )

(defrule desparte_cuvant_in_litere
        ?f <- (word ?s)
        =>
        (retract ?f)  (bind $?w (create$ ))
        (loop-for-count (?i 1 (length ?s))
                (bind $?w (create$ $?w (explode$ (sub-string ?i ?i ?s)))) )
        (assert (word $?w)) )

(defrule V-CV
        ?f <- (word $?first ?v1 ?c ?v2 $?end)
        (test (and (vocala ?v1) (vocala ?v2) (consoana ?c) ))
        (test (or (neq ?v2 i) (neq (length$ $?end) 0) ))                    ; i scurt
        =>
        (retract ?f)  (assert (word $?first ?v1 - ?c ?v2 $?end)) )
```

```
(defrule VC-CV
        ?f <- (word $?first ?v1 ?c1 ?c2 $?med ?v2 $?end)
        (test (and (vocala ?v1) (vocala ?v2) (consoana ?c1) (consoana ?c2) ))
        (test (not (member$ - $?med)))          ;*** pt. execepții "sanc-țiune"***
        =>
        (retract ?f)  (assert (word $?first ?v1 ?c1 - ?c2 $?med ?v2 $?end)) )

(defrule V-CCV_exceptie
        (declare (salience 10))
        (or ?f <- (word $?first ?v1 ?c1&b|c|d|f|g|h|p|t|v  ?c2&l|r $?end)
            ?f <- (word $?first ?v1 ?c1&c|g  ?c2&h $?end))
        (test (vocala ?v1))
        =>
        (retract ?f)  (assert (word $?first ?v1 - ?c1 ?c2 $?end)) )

(defrule VCC-CV_exceptie
        (declare (salience 10))
        ?f <- (word $?first ?v1 &:(vocala ?v1)  ?c1  ?c2  ?c3  $?end)
        (test (str-index(sym-cat ?c1 ?c2 ?c3) "lpt mpt ncs nct ncv dnc rct rtf stm" )))
        =>
        (retract ?f)  (assert (word $?first ?v1 ?c1 ?c2 - ?c3 $?end)) )

(defrule V-D...V-T
        ?f <- (word $?first ?v1 ?v2 ?v3 $?end)
        (test (and (vocala ?v1) (vocala ?v2) (vocala ?v3)))
        (test (or (and (not (triftong ?v1 ?v2 ?v3)) (diftong ?v2 ?v3))
                  (triftong ?v2 ?v3 (nth$ 1 $?end))
                  (not (diftong ?v1 ?v2)) ) )
        =>
        (retract ?f)  (assert (word $?first ?v1 - ?v2 ?v3 $?end)) )

(defrule V-V
        ?f <- (word $?first ?v1 ?v2 $?end)
        (test (and (vocala ?v1) (vocala ?v2) (not (diftong ?v1 ?v2)) ))
        =>
        (retract ?f) (assert (word $?first ?v1 - ?v2 $?end)) )

(defrule hiat...I-A...I-E
        ?f <- (word $?first ?c&:(consoana ?c) ?v1&i ?v2&e|a $?last)
        (test (or (eq ?c d)                              ;precedate de consoana 'd'
                  (eq (length$ $?last) 0) ))             ;ultimele în cuvânt
        =>
        (retract ?f)  (assert (word $?first ?c ?v1 - ?v2 $?last)) )

(defrule afiseaza_cuvant_despartit_in_silabe
        (word $?w)    =>    (printout t (implode$ $?w) t))
```

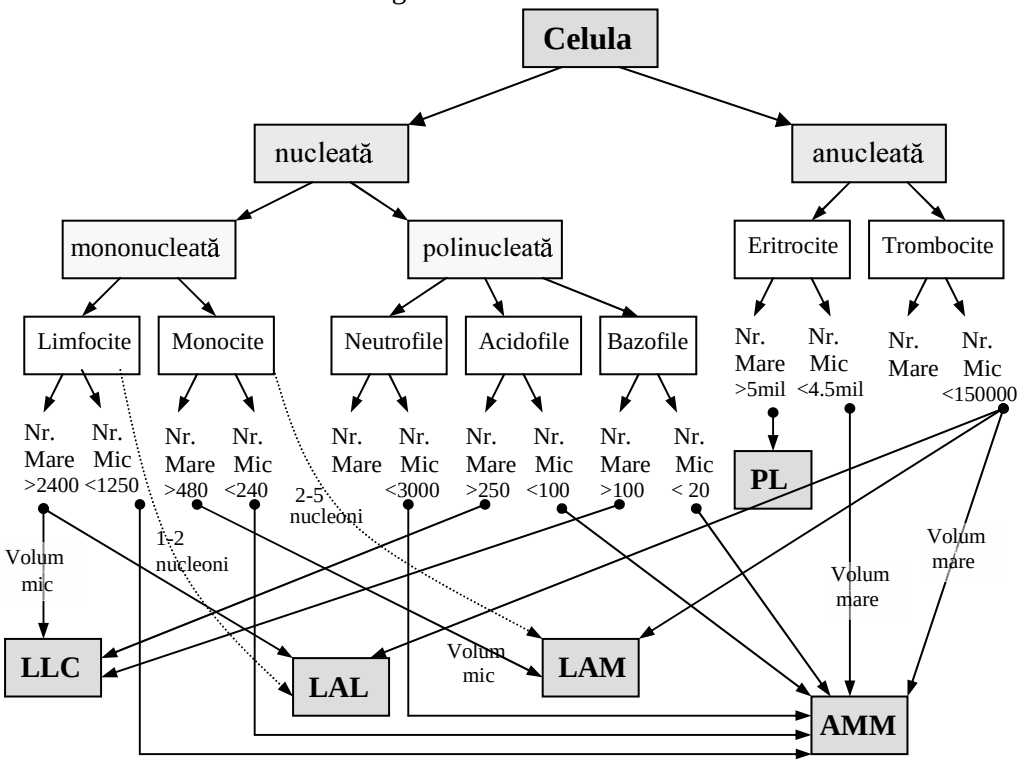
Se observă din nou avantajele oferite de programarea non-procedurală. Codul problemei este foarte uşor de citit şi de înțeles. S-au enumerat practic doar regulile de despărțire în silabe. Pentru vocale în hiat s-au implemetat doar cazurile *I-A* şi *I-E*, fiind necesară şi studierea celorlalte situații.

5.10. Sistem Expert în Medicină

Să presupunem faptul că un program de clasificare a celulelor sanguine determină pentru o imagine microscopică toate tipurile şi numărul aproximativ al celulelor de fiecare tip din proba respectivă şi furnizează programului realizat în CLIPS aceste date. Sistemul Expert trebuie să găsească anomaliile privind prezența unor celule într-un număr mai mare sau mic decât limitele admise şi să ofere un diagnostic. Suplimentar dacă baza de cunoştințe conține şi astfel de informații se pot oferi şi sugestii pentru ameliorarea bolii găsite.

Pentru simplitate programul de față se opreşte doar la diagnosticarea a cinci boli din vastul capitol al patologiei hematologice: leucemia limfatica cronica, leucemia acută mieloblastica, leucemia acută limfoblastica, anemiile macrocitare megaloblastice şi poliglobulia. Nu se oferă informații foarte detaliate legate de forma, culoarea, volumul şi numărul celulelor, deoarece aplicația de față nu îşi propune decât prezentarea modului în care poate fi realizat un astfel de SE. De asemenea pentru sistemul expert avut în vedere, s-au luat în considerație numai acele trăsături care s-au putut " traduce " într-un limbaj de programare.

La o prima clasificare celulele se împart în nucleate (mononucleate şi polinucleate) şi anucleate. Acestea se împart la rândul lor în limfocite, monocite, neutrofile, acidofile, bazofile, eritrocite şi trombocite. În funcție de caracteristicile acestor celule se va oferi un diagnostic.



```
(deftemplate celula  "aceasta este o celula"
        (field cod(type STRING SYMBOL))
        (field diam_cel(type FLOAT)(range 0.0 40.0))
        (field diam_nuc(type INTEGER))
        (field nuclee(type INTEGER))
        (field nucleoli(type FLOAT)(range 0.0 7.0))
        (field vol(type SYMBOL)(allowed-symbols  mare  mic))
        (field frecv(type SYMBOL)(allowed-symbols  mare mica)) )

(deffacts initial
        (celula (cod cel1)(diam_cel 14.5)(nuclee 1)(nucleoli 2.0)(frecv mare))
        (celula (cod cel2)(diam_cel 3.0)(diam_nuc 0)(frecv mica)) )

(defrule R_LAL "regula  pentru leucemie acuta limfoblastica"
        (celula (cod ?c1)(diam_cel ?dc1)(nuclee ?nN1)(nucleoli ?nn1)(frecv ?f1))
        (celula(cod ?c2) (diam_cel ?dc2) (diam_nuc 0) (frecv ?f2))
        (test (and (> ?dc1 12) (< ?dc1 15) (= ?nN1 1) (= ?nn1 2) (eq ?f1 mare)))
        (test (and (> ?dc2 2) (< ?dc2 5) (eq ?f2 mica)))
        =>
        (assert(diagnostic  ?c1  ?c2  LAL))
        (printout t "Diagnostic LAL dedus din celule de tip " ?c1 " si " ?c2 crlf) )

(defrule R_PL "regula pentru poliglobulie"
        (celula (cod ?c3) (diam_cel ?dc3) (diam_nuc 0) (frecv ?f3))
        (test (and (> ?dc3 2) (< ?dc3 7.5) (eq ?f3 mare)) )
        =>
        (assert(diagnostic  ?c3  POLIGLOBULIE))
        (printout t " Diagnostic PL dedus din celule de tip "  ?c3  crlf) )

(defrule R_LAM "regula pentru leucemie acuta mieloblastica"
        (celula (cod ?c4)(diam_cel ?dc4)(nuclee ?nN4)(nucleoli ?nn4)(frecv ?f4))
        (celula (cod ?c2) (diam_cel ?dc2) (diam_nuc 0) (frecv ?f2))
        (test (and (> ?dc4 15) (< ?dc4 25) (= ?nN4 1) (= ?nn4 4) (eq ?f4 mare)))
        (test (and (> ?dc2  2) (<  ?dc2 5) (eq  ?f2  mica)))
        =>
        (assert(diagnostic  ?c4  ?c2  LAM))
        (printout t " Diagnostic LAM dedus din celule de tip " ?c4 " si "  ?c2  crlf) )

; Nu s-a mai studiat şi numărul bazofilelor
(defrule R_LLC "regula pentru leucemie limfoblastica cronica"
        (celula (cod ?c1) (diam_cel ?dc1) (nuclee ?nN1) (frecv ?f1))
        (celula (cod ?c5) (diam_cel ?dc5) (nuclee ?nN5) (frecv ?f5))
        (test (and (> ?dc1 12) (< ?dc1 15) (= ?nN1 1) (eq ?f1 mare)) )
        (test (and (> ?dc5 12) (< ?dc5 15) (> ?nN5 1) (eq ?f5 mare)) )
        =>
        (assert(diagnostic  ?c1  ?c5  ))
        (printout t " Diagnostic LLC dedus din celule de tip " ?c1 " si "  ?c5  crlf))
```

```
(defrule R_AMM "regula pentru anemie megaloblastica macrocitara"
        (celula (cod ?c2) (diam_cel ?dc2) (diam_nuc 0) (vol ?dv2) (frecv ?f2))
        (celula (cod ?c3) (diam_cel ?dc3) (diam_nuc 0) (vol ?dv3) (frecv ?f3))
        (celula (cod ?c1) (diam_cel ?dc1) (nuclee ?nN1) (frecv ?f1))
        (celula (cod ?c4) (diam_cel ?dc4) (nuclee ?nN4) (frecv ?f4))
        (celula (cod ?c5) (diam_cel ?dc5) (nuclee ?nN5) (frecv ?f5))
        (test (and (> ?dc2 2) (< ?dc2 5) (eq ?dv2 mare) (eq ?f2 mica)) )
        (test (and (> ?dc3 2) (< ?dc3 7.5) (eq ?dv3 mare) (eq ?f3 mica)) )
        (test (and (> ?dc1 12) (< ?dc1 15) (= ?nN1 1) (eq ?f1 mica)))
        (test (and (> ?dc4 15) (< ?dc4 25) (= ?nN4 1) (eq ?f4 mica)) )
        (test (and (> ?dc5 12) (< ?dc5 15) (> ?nN5 1) (eq ?f5 mica)))
        =>
        (assert(diagnostic  ?c2  ?c3  ?c1  ?c4  ?c5  AMM))
        (printout t "Diagnostic AMM dedus din celule " ?c2 , ?c3 , ?c1 , ?c4 " si " ?c5 t))

(defrule R_LIPSA_DIAGNOSTIC
        (not (diagnostic $?))
        =>
        (printout t "Din  celulelor analizate nu se poate diagnostica o boala !" t))
```

În regulile de mai sus au fost neglijate bazofilele, deoarece acestea au un coeficient de importanță foarte mic în stabilirea diagnosticului. Dacă se dorește se poate înlocui în faptele inițiale valorile *cel1, cel2* ale sloturilor tip cu simbolurile limfocite şi trombocite. În locul variabilelor *?c1*, *?c2* ... *?c5* se pot pune direct numele celor 5 tipuri de celule dar se pierde astfel din flexibilitatea programului. Numerele de ordine folosite în program pentru fiecare tip de celulă sunt (1)limfocite (2) trombocite, (3) eritrocite, (4) monocite, (5) neutrofile.

Pentru valorile sloturilor frecvență şi volum se puteau defini în varianta FuzzyCLIPS variabile fuzzy, pentru fiecare fapt şi fiecare regulă se puteau asocia coeficienți de certitudine, şi astfel în final faptul diagnostic ar fi avut un coeficient de certitudine CF, mai apropiat de valoarea reală. Aceşti coeficienți asigură programului care interpretează imaginile medicale, posibilitatea de a oferi în cazul în care anumite zone nu au putut fi analizate şi un număr (exprimat în procente) care să specifice pe "cât la sută" suprafața imaginii au fost identificate celule. Acest număr obținut în urma procesului de defuzificare poate fi modelat cu ajutorul unor operatori lingvistici de genul "foarte sigur", "cert" , "probabil", "puțin probabil" pentru a ne apropia de modul de exprimarea a diagnosticului în cazul în care acesta ar fi fost oferit de un specialist.